AEDII - Atividade 1 Nome: Bernardo Carvalho

Questão 1: Ponto de Eficiência
A partir de quantos valores (n) se mostra mais eficiente buscar elementos utilizando uma estrutura de árvore binária ao invés de uma lista? Prove em código sua resposta.

A superioridade da busca em uma árvore binária já pode ser percebida com poucos valores, com um *n* de 100, por exemplo, a árvore termina sua busca em 0.000027s, enquanto uma lista encadeada demora 0.000181s. Como sabemos, valores muito pequenos tendem a ser mais suscetíveis a oscilações advindas de outros fatores que não do próprio algoritmo empregado, como variações no tempo de execução da linguagem e/ou questões de *cache*. Para mitigar tais efeitos, foram realizados testes (no arquivo em anexo) com diferentes valores de *n*, como 100, 1.000, 100.000 e 5.000.000 de modo a demonstrar a evolução dos tempos de busca dos algoritmos de acordo com o tamanho da amostra. A partir de 1.000, por exemplo, a árvore consegue terminar a busca quase que instantaneamente, enquanto a busca na lista já demora quase 1 segundo. Vale ressaltar que, com intuito de acelerar a criação da lista, foi realizada uma otimização na estrutura, permitindo que esta armazene, além da raíz, o último nó inserido. Isso possibilita que nós adicionais sejam ligados diretamente ao último da lista. Sem tal otimização, não seria possível realizar a criação da lista com o conjunto grande por questões de tempo de execução.

Questão 2: Análise Temporal Detalhada
Documente os tempos parciais para cada conjunto de dados.

Busca por valores que existem e não existem
n=1000, seed_criacao=50, seed_busca=42
Tempo construção árvore: 0.000262
Tempo busca árvore: 0.000275
Tempo de construção da lista: 0.007678
Tempo de busca da lista: 0.010894

n=100000, seed_busca=42, arquivo=conjunto_pequeno.txt
Tempo construção árvore: 0.077107
Tempo busca árvore: 0.000052

Tempo de construção da lista: 72.850588
Tempo de construção da lista: 0.010250 (otimização inserção)
Tempo de busca da lista: 0.126146

n=5000000, seed_busca=42, arquivo=conjunto_medio.txt
Tempo construção árvore: 11.242179
Tempo busca árvore: 0.000102
Tempo de construção da lista: 0.989898 (otimização inserção)
Tempo de busca da lista: 6.780271

n=30000000, seed_busca=42, arquivo=conjunto_grande.txt
Tempo construção árvore: 93.391699
Tempo busca árvore: 0.000078
Tempo de construção da lista: 5.815195 (otimização inserção)
Tempo de busca da lista: 36.940434

Busca por valores que não existem na lista
n=100000, valores_busca=[11, 21, 27, 28, 29, 30, 32, 445], arquivo=conjunto_pequeno.txt
Tempo construção árvore: 0.066696
Tempo busca árvore: 0.000010
Tempo de construção da lista: 0.011005 (otimização inserção)
Tempo de busca da lista: 0.011278

n=5000000, valores_busca=[11, 21, 27, 28, 29, 30, 32, 445], arquivo=conjunto_medio.txt
Tempo construção árvore: 10.894585
Tempo busca árvore: 0.000016
Tempo de construção da lista: 0.977510 (otimização inserção)
Tempo de busca da lista: 0.538754

n=30000000, valores_busca=[11, 21, 27, 28, 29, 30, 32, 445], arquivo=conjunto_grande.txt
Tempo construção árvore: 87.310996
Tempo busca árvore: 0.000023
Tempo de construção da lista: 5.400947 (otimização inserção)
Tempo de busca da lista: 3.289316

Questão 3: Análise de Ineficiências
As listas se mostram mais lentas para grandes conjuntos durante a busca por valores? Se são ineficientes, explique detalhadamente os motivos técnicos dessa ineficiência.

Sim, quanto maior o conjunto utilizado, maior foi a diferença de tempo entre a busca em uma árvore binária e em uma lista. O motivo dessa ineficiência é conceitual, e decorre da maneira com que ambas estruturas de dados realizam a busca. Como a árvore binária organiza o conjunto no momento da construção da estrutura, ela acaba por otimizar a posterior busca, visto que esta tende a dividir o espaço de busca pela metade. A lista, por outro lado, não realiza nenhuma ordenação no momento de inserção de valores, o que faz com que a busca tenha de ser realizada elemento por elemento, dado que não se sabe se o valor do nó seguinte é maior ou menor do que o valor atual. Desta forma, na média, os valores são encontrados mais rapidamente na árvore do que na lista. Apesar destas diferenças, algumas observações devem ser realizadas. A árvore desenvolvida não possui balanceamento, de modo a que a inserção de dados em uma ordem específica (como 1, 2, 3, 4) pode acabar formando uma árvore que tem comportamento idêntico a uma lista (visto que todos os nós estarão à esquerda ou direita do nó anterior, essencialmente como uma lista). Além disso, se considerarmos os piores casos possíveis, como quando buscamos um valor que está no último nó ou um valor que não está em nenhuma das estruturas, ambas têm de percorrer todos os nós, de modo a que podemos afirmar que, apesar da busca na árvore ser mais rápida na média, as duas possuem a mesma complexidade, O(n).

Questão 4: Otimização de Listas
É possível otimizar o algoritmo da Classe Lista criada, sem deixar de operar em uma LISTA, para melhorar a performance de busca?

Com o objetivo de manter a implementação original (e a otimização inicial que havia feito) da classe *Lista*, foi criada uma nova classe chamada *ListaQuestao4* para responder esta questão, devidamente documentada nas seções posteriores**.** Essa nova lista busca sacrificar o ganho no tempo de inserção obtido na otimização da classe *Lista* original para acelerar a busca posterior. Para tal, toda vez que um novo valor for inserido, a lista já selecionará o melhor local para este, de modo a manter uma ordem crescente. Isso acelera

significativamente a busca, visto que ao percorrer a lista, caso o valor do nó atual seja maior que o valor buscado, já é possível assumir que o valor não está na lista. Segue os resultados:

n=1000, seed_criacao=50, seed_busca=42
Tempo de construção da lista nova: 0.005048
Tempo de construção da lista anterior: 0.007678 (sem a otimização anterior)
Tempo de busca da lista nova: 0.012138
Tempo de busca da lista: 0.010894

n=100000, seed_busca=42, arquivo=conjunto_pequeno.txt
Tempo de construção da lista nova: 56.735777
Tempo de construção da lista anterior: 72.850588 (sem a otimização anterior)
Tempo de construção da lista anterior: 0.010250 (com otimização anterior)
Tempo de busca da lista nova: 0.000158
Tempo de busca da lista anterior: 0.126146

n=100000, valores_busca=[11, 21, 27, 28, 29, 30, 32, 445], arquivo=conjunto_pequeno.txt
Tempo de construção da lista nova: 53.502623
Tempo de construção da lista anterior: 0.011005 (otimização inserção)
Tempo de busca da lista nova: 0.000008
Tempo de busca da lista anterior: 0.011278

n=10000, seed_criacao=50, seed_busca=42
Tempo de construção da lista nova: 0.471865
Tempo de busca da lista nova: 1.160095
Tempo de construção da lista antiga: 0.728615 (sem a otimização anterior)
Tempo de busca da lista antiga: 0.178855
Como pôde ser observado, as “melhorias” foram muito inconsistentes, de modo a que seria necessária uma melhor otimização desse algoritmo. Não foram realizados testes com conjuntos maiores, pois a nova lista demorou muito para ser criada já ordenada, impossibilitando a gravação dos tempos.

Explicação detalhada das decisões algorítmicas tomadas e justificativa para a criação de cada classe e método

Classe Utils: Contém métodos auxiliares para as principais classes do projeto.

- *ler_arquivo*: Realiza a leitura do conteúdo de um arquivo e retorna o conteúdo em formato de lista.
- *contar_items*: Método auxiliar substituto do método *len*.

Classe NodeLista: Contém os atributos básicos de um nó de uma lista ligada, como valor e o próximo nó.

Classe NodeArvore: Contém os atributos básicos de um nó de uma árvore binária, como valor e os nós seguintes, à esquerda e à direita.

Classe Arvore: Implementa uma árvore binária.

- *construtor*: Inicializa a raiz para None.

- *inserir (valor)*:
    - verifica se existe raiz na árvore atual:
        - caso não exista, cria um novo nó com o valor a ser inserido e adiciona ele como raiz;
        - caso exista, verifica se o valor a ser inserido na lista é menor que o do nó atual (a própria raiz, na primeira iteração):
            - caso seja, verifica se existe algum valor à esquerda do nó atual:
                - caso não exista, cria-se um novo nó com o valor a ser inserido à esquerda do nó atual.
                - caso exista, o nó atual passa a ser o nó que está à esquerda do atual
            - caso não seja, verifica se o valor a ser inserido na lista é maior que o do nó atual (a própria raiz, na primeira iteração)
        - por fim, se o valor a ser inserido não é maior nem menor que o do nó atual, logo ele é igual, portanto não fazemos nada.

- *buscar (valor) -> bool*:
    - atribui o nó atual ao nó raiz:

        - caso não exista, logo a lista está vazia, portanto o valor não está na lista;
        - enquanto o nó atual for diferente de nulo:
            - caso o valor buscado seja igual ao valor do nó atual, o valor foi encontrado:
            - caso o valor buscado seja menor que o valor do nó atual, tornamos o nó à esquerda do atual o novo nó atual e continuamos procurando:
            - caso o valor buscado seja maior que o valor do nó atual, tornamos o nó à direita do atual o novo nó atual e continuamos procurando:
        - por fim, se o valor buscado ainda não foi encontrado, logo ele não está na árvore, pois sua totalidade foi percorrida.

Classe Lista: Implementa uma lista ligada.

- *construtor*: Inicializa a raiz e o último nó para None e recebe o parâmetro otimizada, para verificar se gostaria de usar a lista com inserção e busca otimizadas (padrão False).

- *inserir (valor)*:
    - verifica se a lista atual é otimizada:
        - caso seja, verifica se existe raiz na lista atual:
            - caso não exista, cria um novo nó com o valor a ser inserido e adiciona ele como raiz e último nó.
            - caso exista, coloca o último nó ao nó atual:
                - cria um novo nó com o valor a ser inserido e o coloca como o próximo nó como nó atual
        - caso não seja otimizada, também verifica se existe raiz na lista atual:
            - caso não exista, cria um novo nó com o valor a ser inserido apenas.
            - caso exista, coloca o nó raiz como nó atual:
                - caso o nó atual não aponte para nenhum outro, cria-se um nó com o valor a ser inserido e atribui este nó como o próximo do nó atual.
                - caso o nó atual aponte para outro, torna o nó que ele aponta como o nó atual.

- *buscar (valor) -> bool*:
    - atribui o nó atual ao nó raiz:
        - caso não exista, logo a lista está vazia, portanto o valor não está na lista;
        - enquanto o nó atual for diferente de nulo:
            - caso o valor do nó atual seja igual ao valor buscado, o valor foi encontrado na lista:
            - caso contrário, torna o nó que ele aponta como o nó atual:
        - por fim, se o valor buscado ainda não foi encontrado, logo ele não está na lista, pois sua totalidade foi percorrida.

Classe ListaQuestao4: Implementa uma lista ligada “otimizada”.

- *construtor*: Inicializa a raiz da lista como None, criando uma lista inicialmente vazia.

- *inserir (valor)*:
    - cria um novo nó com o valor a ser inserido:
        - verifica se a lista atual está vazia:
            - caso esteja, insere o novo nó como raiz da lista e finaliza o processo.
            - caso contrário, verifica se o valor a ser inserido deve ficar antes da raiz:
                - se sim, o novo nó aponta para a raiz atual, e ele passa a ser a nova raiz da lista.
            - caso o valor não seja menor que a raiz, percorre a lista a partir da raiz até encontrar a posição correta:
                - enquanto o próximo nó existir e tiver valor menor que o valor a ser inserido, o nó atual avança para o próximo.
                - ao encontrar a posição correta, o novo nó aponta para o próximo nó do atual, e o atual passa a apontar para o novo nó, inserindo-o no meio ou final da lista.

- *buscar (valor) -> bool*:
    - atribui o nó atual ao nó raiz:

    - caso não exista, logo a lista está vazia, portanto o valor não está na lista;
    - enquanto o nó atual for diferente de nulo:
        - caso o valor do nó atual seja igual ao valor buscado, o valor foi encontrado na lista.
        - caso o valor do nó atual seja maior que o valor buscado, retorna False, pois a lista está ordenada e o valor não pode mais aparecer à frente.
        - caso contrário, passa para o próximo nó.
    - por fim, se o valor buscado ainda não foi encontrado, logo ele não está na lista, pois sua totalidade foi percorrida.

Métodos avulsos: Utilizados para automatizar os testes.

- *testar_arvore(n=100, valores_busca=None, seed_criacao=50, seed_busca=42, nome_arquivo=None)*: Cria, popula e busca em uma árvore binária de acordo com os parâmetros informados, além de realizar as marcações de tempo de cada atividade.

- *testar_lista(n=100, valores_busca=None, seed_criacao=50, seed_busca=42, nome_arquivo=None, otimizada=False)*: Cria, popula e busca em uma lista ligada de acordo com os parâmetros informados, além de realizar as marcações de tempo de cada atividade. Possui a possibilidade de rodar uma versão otimizada da lista, cuja inserção e busca são mais velozes.

- *testar_lista_questao_4(n=100, valores_busca=None, seed_criacao=50, seed_busca=42, nome_arquivo=None)*: Cria, popula e busca em uma lista ligada “otimizada” de acordo com os parâmetros informados, além de realizar as marcações de tempo de cada atividade.

Análise dos problemas enfrentados e soluções implementadas e discussão sobre as vantagens e desvantagens de cada estrutura

O primeiro problema enfrentado foi a demora para criar listas ligadas muito extensas. No caso do conjunto grande, por exemplo, foi impossível criá-la em um tempo aceitável. Após pesquisar, percebi que a implementação inicial da lista estava realmente de acordo com o conceito clássico, porém o tempo de criação

era inaceitável. Surgiu então a primeira melhoria, onde a lista passou a gravar seu último nó. Essa otimização fez com que a lista fosse montada mais rápido do que a árvore em praticamente todos os testes, permitindo que os testes de busca com conjuntos maiores fossem possíveis de realizar com maior agilidade.

A respeito das vantagens e desvantagens, para conjuntos muito pequenos, mesmo sem otimização, as listas ligadas cumprem bem seu papel, todavia, as árvores binárias tendem a ser mais confiáveis na questão da escalabilidade, ainda mais se forem implementadas melhorias, como o balanceamento da árvore, por exemplo, que garante uma complexidade de O(log n). Como a árvore criada aqui não possui este recurso, sua complexidade pode ser, no pior dos casos, a mesma da lista ligada O(n), porém, como observado nos testes, ainda assim ela tende a ser mais veloz, na média, do que a lista ligada quando a questão é busca.

Por fim, gostaria de mencionar o maior problema de todos: a “otimização” da lista. Como demonstrado na questão 4, não foi possível melhorar a busca da lista ligada de maneira consistente, mesmo sacrificando o tempo de inserção anteriormente acelerado. Obviamente, deve existir alguma maneira de realizar esta “melhoria” de maneira consistente e mais correta, porém realmente não foi possível desenvolvê-la, por ora.

Sistema utilizado nos testes:
CPU: Ryzen 7 9700X em stock.
RAM: 32GB DDR5@6400MHz.
OS: Windows 11 Pro.